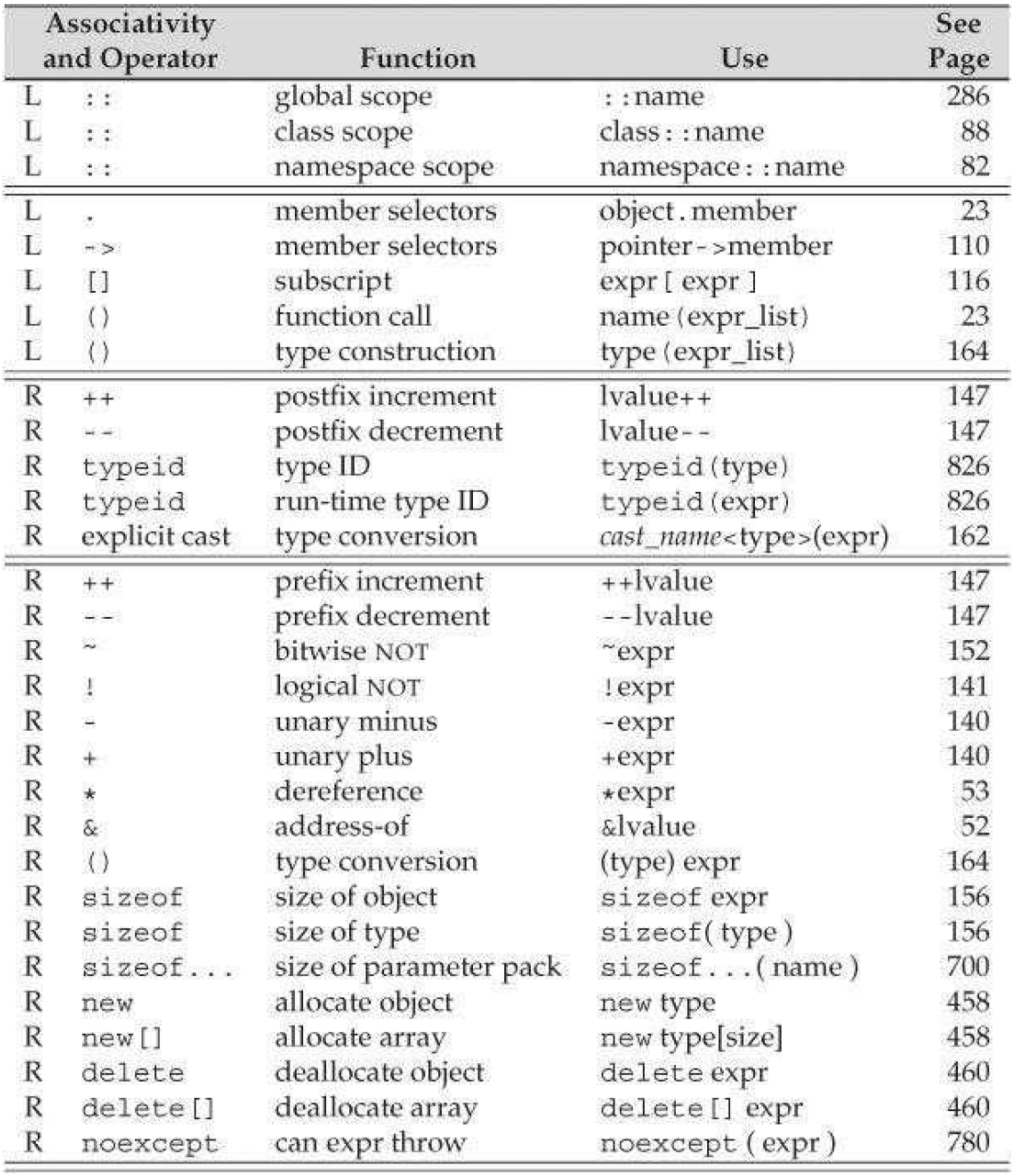
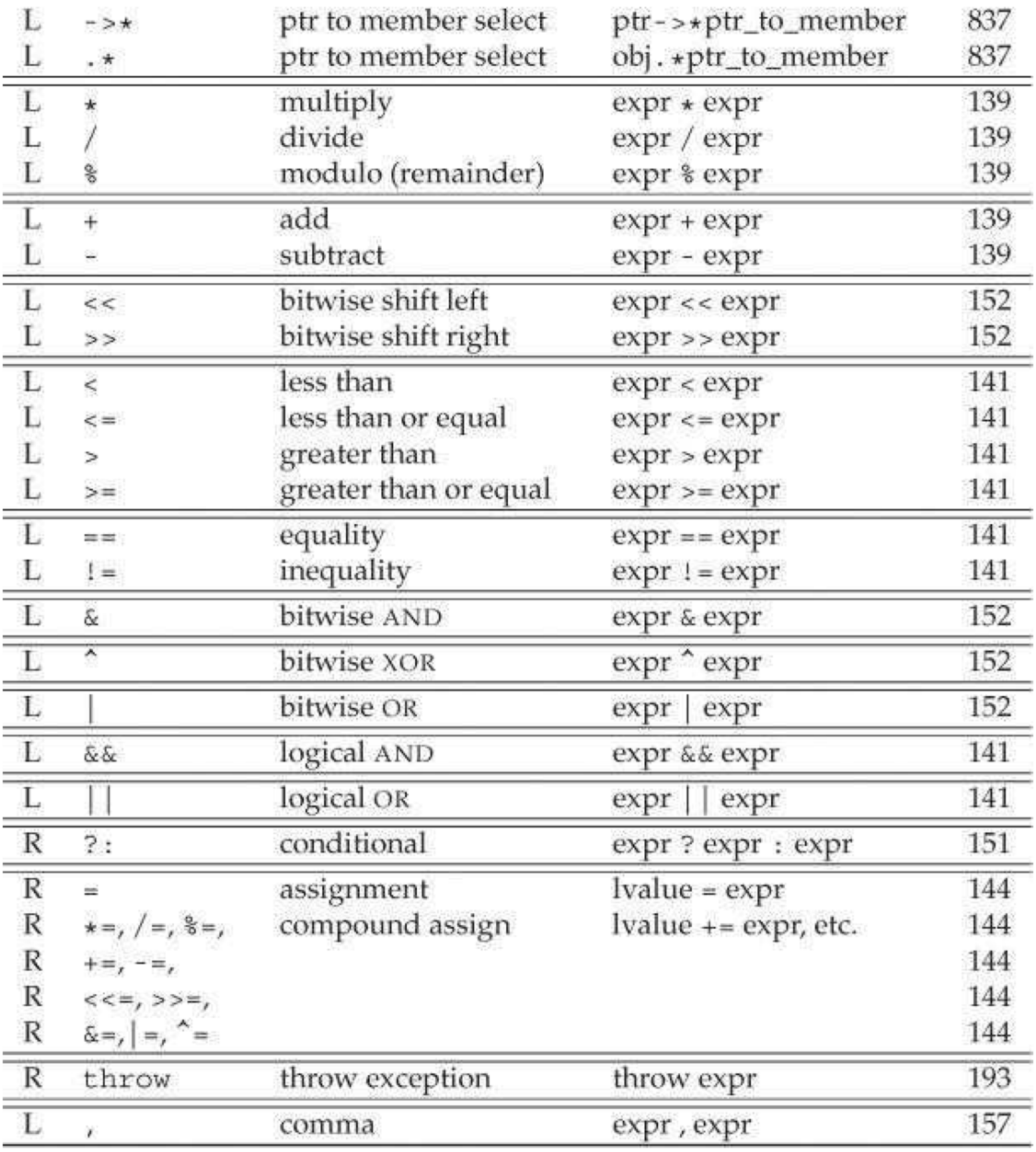
Table 4.4. Operator Precedence



Associativity and Operator		Function	Use	See Page
L	::	global scope	::name	286
L	::	class scope	class::name	88
L	::	namespace scope	namespace::name	82
L	.	member selectors	object.member	23
L	->	member selectors	pointer->member	110
L	[]	subscript	expr[expr]	116
L	()	function call	name(expr_list)	23
L	()	type construction	type(expr_list)	164
R	++	postfix increment	lvalue++	147
R	--	postfix decrement	lvalue--	147
R	typeid	type ID	typeid(type)	826
R	typeid	run-time type ID	typeid(expr)	826
R	explicit cast	type conversion	*cast_name*<type>(expr)	162
R	++	prefix increment	++lvalue	147
R	--	prefix decrement	--lvalue	147
R	~	bitwise NOT	~expr	152
R	!	logical NOT	!expr	141
R	-	unary minus	-expr	140
R	+	unary plus	+expr	140
R	*	dereference	*expr	53
R	&	address-of	&lvalue	52
R	()	type conversion	(type) expr	164
R	sizeof	size of object	sizeof expr	156
R	sizeof	size of type	sizeof(type)	156
R	sizeof...	size of parameter pack	sizeof...(name)	700
R	new	allocate object	new type	458
R	new[]	allocate array	new type[size]	458
R	delete	deallocate object	delete expr	460
R	delete[]	deallocate array	delete[] expr	460
R	noexcept	can expr throw	noexcept (expr)	780



L	->*	ptr to member select	ptr->*ptr_to_member	837
L	.*	ptr to member select	obj.*ptr_to_member	837
L	*	multiply	expr * expr	139
L	/	divide	expr / expr	139
L	%	modulo (remainder)	expr % expr	139
L	+	add	expr + expr	139
L	-	subtract	expr - expr	139
L	<<	bitwise shift left	expr << expr	152
L	>>	bitwise shift right	expr >> expr	152
L	<	less than	expr < expr	141
L	<=	less than or equal	expr <= expr	141
L	>	greater than	expr > expr	141
L	>=	greater than or equal	expr >= expr	141
L	==	equality	expr == expr	141
L	!=	inequality	expr != expr	141
L	&	bitwise AND	expr & expr	152
L	^	bitwise XOR	expr ^ expr	152
L	|	bitwise OR	expr | expr	152
L	&&	logical AND	expr && expr	141
L	||	logical OR	expr || expr	141
R	?:	conditional	expr ? expr : expr	151
R	=	assignment	lvalue = expr	144
R	*=, /=, %=,	compound assign	lvalue += expr, etc.	144
R	+=, -=,			144
R	<<=, >>=,			144
R	&=, |=, ^=			144
R	throw	throw exception	throw expr	193
L	,	comma	expr , expr	157

6.2.2 Passing Arguments by Reference

Example: return the index of the first occurrence of char in a string

```
// returns the index of the first occurrence of c in s
// the reference parameter occurs counts how often c occurs
string::size_type find_char(const string &s, char c,
   string::size_type &occurs)
{
   auto ret = s.size(); // position of the first occurrence, if any
   occurs = 0; // set the occurrence count parameter
   for (decltype(ret) i = 0; i != s.size(); ++i) {
      if (s[i] == c) {
         if (ret == s.size())
            ret = i; // remember the first occurrence of c
         ++occurs; // increment the occurrence count
      }
   }
   return ret; // count is returned implicitly in occurs
}
```

We can then call the `find_char` as follows:

```
string s = “Hello world”;
string::size_type cnt;
auto index find_char(s, ‘o’, cnt);
```

Quick Check: What is the value of `index` and `cnt` after the function call?

A:

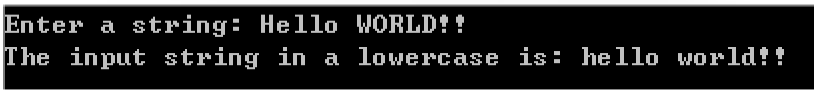
Exercise 6.1 In-class Coding Exercise

Ex61.cpp

Write a function `stringToLower` to change a given string to all lowercase.

```
#include <iostream>
#include <string>
#include <cctype>
using namespace std;


(YOUR FUNCTION HERE)
```

```
int main()
{
   string s ;
   cout << "Enter a string: ";
   getline(cin, s);
   cout << "The input string in a lowercase is: "
        << stringToLower(s) << endl;
   return 0;
}
```



```
Enter a string: Hello WORLD!!
The input string in a lowercase is: hello world!!
```

Answer:

6.2.4 Array Parameters

(see ppt)

A function can have a parameter for an entire array so that when the function is called, the argument that is plugged in for this formal parameter is an entire array. However, a parameter for an entire array is neither a call-by-value parameter nor call-by-reference parameter. It is a new kind of formal parameter referred to as an **array parameter**.

```
void fillUp(int a[], size_t size)
{
   // fillUp the array …
}
```

The parameter `int a[]` is an array parameter and `size_t` is a machine-specific unsigned type that is large enough to hold the size of any object in memory. In `int a[],` the empty square brackets with no index expressed inside, are what C++ uses to indicate an array parameter. An array parameter is not quite a call-by-reference parameter but for most practical purposes, it behaves very much like a call-by-reference parameter. When passing the array as an argument, you simply pass the name of array (the memory address).

```
    ...
    int a[20];
    fillUp(a, 20);
    ...
```

When an array argument is plugged in for an array parameter, all that is given to the function is the address in memory of the first element (the one indexed by zero). In C++, when we use **the name of an array** in an expression, that name is automatically converted into **the memory address of the first element of the array.**

Because arrays are converted to pointers, when we pass an array to a function, we are actually passing a pointer to the array's first element. Thus,

```
void fillUp(int a[], size_t size);
```

```
void fillUp(int* a, size_t size);
```

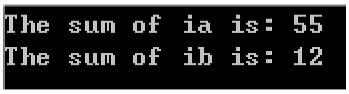
are equivalent.

Exercise 6.2 In-class Coding Exercise

Ex62.cpp

Write a `sum` function that sums up all elements in an `int` array and return the sum.

```
#include <iostream>

using namespace std;


(YOUR FUNCTION HERE)





int main()
{
    int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int ib[3] = {1, 3, 8};
    cout << "The sum of ia is: " << sum(ia, 10) << endl;
    cout << "The sum of ib is: " << sum(ib, 3) << endl;
    return 0;
}
```



```
The sum of ia is: 55
The sum of ib is: 12
```

Answer:

Because arrays are passed as pointers, functions ordinarily **DO NOT** know the size of the array they are given. They must rely on additional information provided by the caller. We can explicitly pass a size parameter as we did in the previous exercise. It can also be achieved by `begin` and `end` iterators:

Exercise 6.3 In-class Coding Exercise

Alternative solution using `begin` and end `iterators`

Ex63.cpp

```
#include <iostream>
#include <iterator>

using namespace std;

(YOUR FUNCTION HERE)







int main()
{
   int ia[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int ib[3] = {1, 3, 8};
   cout << "The sum of ia is: " << sum(begin(ia), end(ia)) << endl;
   cout << "The sum of ib is: " << sum(begin(ib), end(ib)) << endl;
   return 0;
}
```

Answer:

6.3 Return Types

(see ppt)

Return non-reference and reference types

```
// return plural version of word if ctr isn't 1
string make_plural(size_t ctr, const string &word,
      const string &ending)
{
    return (ctr == 1) ? word : word + ending;
}
```

```
// find longer of two strings
const string &shorterString(const string &s1, const string &s2)
{
    return s1.size() < s2.size() ? s1 : s2;
}
```

Now let's look at a **terribly incorrect** program; **what's wrong?**

```
const string& manip(const string& s)
{
     string ret = s;
     ret += “ is terrible”;
     return ret;
}
```

A: